Linux Basics (II)

Healthcare Data Science (BIOS 511)

- Secure shell (SSH)
    - SSH
    - Transfer files between machines
- Run R in Linux
    - Interactive mode
    - Batch mode
    - Pass arguments to R scripts
    - Run long jobs
    - Large simulation studies

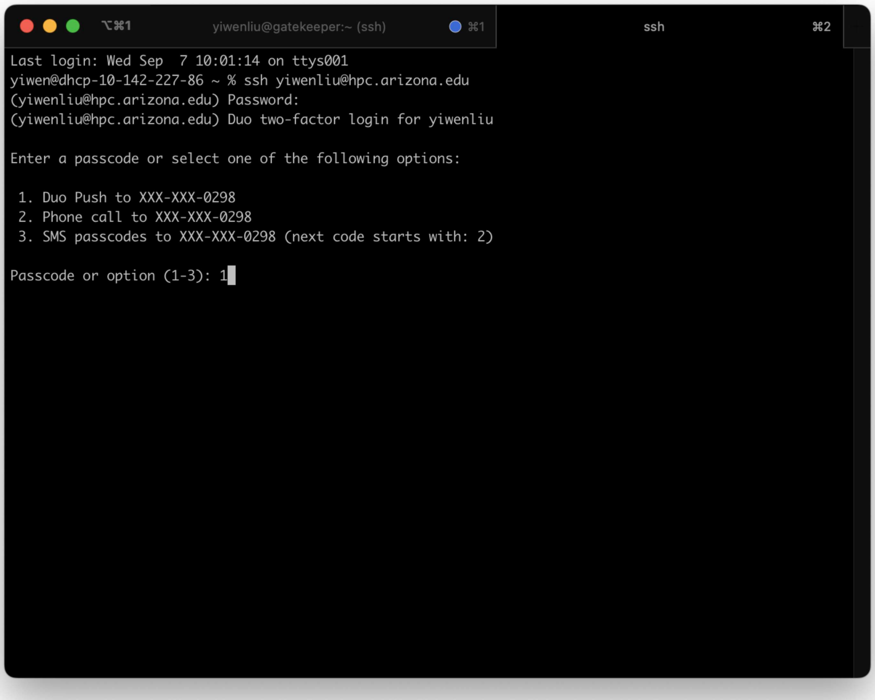
Secure shell (SSH)

SSH

SSH (secure shell) is the dominant cryptographic network protocol for secure network connection via an insecure network.

- On Linux or Mac, access the server by

```
ssh netid@hpc.arizona.edu
```



```
Last login: Wed Sep  7 10:01:14 on ttys001
yiwen@dhcp-10-142-227-86 ~ % ssh yiwenliu@hpc.arizona.edu
(yiwenliu@hpc.arizona.edu) Password:
(yiwenliu@hpc.arizona.edu) Duo two-factor login for yiwenliu

Enter a passcode or select one of the following options:

 1. Duo Push to XXX-XXX-0298
 2. Phone call to XXX-XXX-0298
 3. SMS passcodes to XXX-XXX-0298 (next code starts with: 2)

Passcode or option (1-3): 1
```

- Windows machines need the PuTTY (http://www.putty.org) program (free) or Terminus (https://termius.com/windows).

Transfer files between machines

- `scp` securely transfers files between machines using SSH.

```
## copy file from local to remote
scp localfile username@filexfer.hpc.arizona.edu:/path_to_folder
```

```
## copy file from remote to local
scp username@filexfer.hpc.arizona.edu:/path_to_file pathtolocalfolder
```

- GUIs for Windows (WinSCP) or Mac (Cyberduck).
- Use a **version control system** to sync project files between different machines and systems.

Run R in Linux

Interactive mode

- Start R in the interactive mode

```
module load R
R
```

- Then run R script by

```
source("script.R")
```

Batch mode

- Demo script `meanEst.R` (./meanEst.R?_&d2lSessionVal=msHLP93GphIxJVZ4ex9VUCmde&ou=1203592) implements an (terrible) estimator of mean

$$\widehat{\mu}_n = \frac{\sum_{i=1}^{n} x_i 1_{i \text{ is prime}}}{\sum_{i=1}^{n} 1_{i \text{ is prime}}}.$$

```
## ## check if a given integer is prime
## isPrime = function(n) {
##   if (n <= 3) {
##     return (TRUE)
##   }
##   if (any((n %% 2:floor(sqrt(n))) == 0)) {
##     return (FALSE)
##   }
##   return (TRUE)
## }
##
## ## estimate mean only using observation with prime indices
## estMeanPrimes = function (x) {
##   n = length(x)
##   ind = sapply(1:n, isPrime)
##   return (mean(x[ind]))
## }
##
## print(estMeanPrimes(rnorm(100000)))
```

- To run your R code non-interactively aka in batch mode, we have at least two options:

```
# default output to meanEst.Rout
R CMD BATCH meanEst.R
```

or

```
# output to stdout
Rscript meanEst.R
```

```
## [1] 0.01625258
```

- Typically automate batch calls using a scripting language, e.g., Python, perl, and shell script.

Pass arguments to R scripts

When using the command line to run an R script file, we may want to pass arguments and save the output in a file. We demonstrates these techniques using the `R CMD BATCH` command and the `Rscript` front-end. In general:

```
R CMD BATCH [options] infile [outfile]
```

Motivating question

We intend to use the R code below to randomly generate a value from the normal distribution with given parameters and random seed.

```
cat script.R
```

```
## myfunction=function(seed, mu, sig){
##   set.seed(seed)
##   return( rnorm(1,mu, sig) )
## }
## seed=1234
## mu=0
## sig=1
## print(myfunction(seed, mu, sig))
```

```
R CMD BATCH script.R
```

- Step I. Determine your input/arguments: `seed` , `mu` , and `sig` .
- Step II. Arguments passed on the command line should be retrieved using the `commandArgs()` function. Modify the R code

  ```
  cat arg_script.R
  ```

  ```
  ## myfunction=function(seed, mu, sig){
  ##   set.seed(seed)
  ##   return( rnorm(1, mu, sig) )
  ## }
  ##
  ## ## Get the arguments
  ## args=commandArgs(trailingOnly = TRUE)
  ## args
  ##
  ## seed=as.numeric(args[1]) # The first argument
  ## mu=as.numeric(args[2])
  ## sig=as.numeric(args[3])
  ## print(myfunction(seed, mu, sig))
  ```

- Step III. Specify arguments in `R CMD BATCH` :

  ```
  R CMD BATCH '--args 1234 0 1' arg_script.R
  ```

- Or specify arguments in `Rscript` :

  ```
  Rscript arg_script.R 1234 0 1 > Routput.txt
  ```

IMPROVE: how about this?

```
R CMD BATCH '--args seed=1234 mu=0 sig=1' arg_script.R
```

- `Parse` command line arguments using `commandArgs` . This function scans the arguments which have been supplied when the current R session was invoked. To understand the magic formula `parse` and `eval` :

  ```
  rm(list=ls())
  print(x)
  ```

```
## Error in print(x): object 'x' not found
```

```
parse(text="x=3")
```

```
## expression(x = 3)
```

```
eval(parse(text="x=3"))
print(x)
```

```
## [1] 3
```

- If you have arguments that needs to be passed to R, include:

```
for (arg in commandArgs(T)) {
  eval(parse(text=arg))
}
```

After calling the above code, all command line arguments will be **available in the global namespace**.

Practice

- Modify `meanEst.R` (./meanEst.R?_&d2lSessionVal=msHLP93GphIxJVZ4ex9VUCmde&ou=1203592) to include arguments `seed` , `distr` (distribution) and `n` (number of data values to be generated). When `dist="gaussian"` , generate data from standard normal distribution (`rnorm`), when `dist="poisson"` , generate data from Poisson distribution (`rpois`) with `lambda=1` .
- original code

```
## ## check if a given integer is prime
## isPrime = function(n) {
##   if (n <= 3) {
##     return (TRUE)
##   }
##   if (any((n %% 2:floor(sqrt(n))) == 0)) {
##     return (FALSE)
##   }
##   return (TRUE)
## }
##
## ## estimate mean only using observation with prime indices
## estMeanPrimes = function (x) {
##   n = length(x)
##   ind = sapply(1:n, isPrime)
##   return (mean(x[ind]))
## }
##
## print(estMeanPrimes(rnorm(100000)))
```

- Modify the code so that the following command can be implemented

```
R CMD BATCH '--args seed=1234 dist="gaussian" n=10' arg_meanEst.R
```

or

```
Rscript arg_meanEst.R seed=1234 'dist="gaussian"' n=10 > output.txt
```

- The modified code

```
cat arg_meanEst.R
```

```
## ## check if a given integer is prime
## isPrime = function(n) {
##   if (n <= 3) {
##     return (TRUE)
##   }
##   if (any((n %% 2:floor(sqrt(n))) == 0)) {
##     return (FALSE)
##   }
##   return (TRUE)
## }
##
## ## estimate mean only using observation with prime indices
## estMeanPrimes = function (x) {
##   n = length(x)
##   ind = sapply(1:n, isPrime)
##   return (mean(x[ind]))
## }
##
## for(i in 1:length(commandArgs(TRUE))){
##   eval(parse(text=commandArgs(TRUE)[i]))
## }
##
## set.seed(seed)
## if(dist=="gaussian"){
##   print(estMeanPrimes(rnorm(n)))
## } else if(dist=="poisson"){
##   print(estMeanPrimes(rpois(n,lambda=1)))
## } else {
##   print("Choose distributions between gaussian and poisson")
## }
```

Run long jobs

Many statistical computing tasks take long: simulation, MCMC, etc.

- `nohup` command in Linux runs program(s) immune to hangups and writes output to `nohup.out` by default. Logging out will *not* kill the process; we can log in later to check status and results. It is POSIX standard thus available on Linux and MacOS. For example, run `runSim.R` in background and writes output to `nohup.out`:

```
nohup Rscript arg_meanEst.R seed=1234 'dist="poisson"' n=100
```

```
## [1] 0.5
```

- `screen` is another popular utility, but not installed by default. Typical workflow using `screen`.

   0. Access remote server using `ssh`.

   1. Start jobs in batch mode.

   2. Detach jobs.

   3. Exit from server, wait for jobs to finish.

   4. Access remote server using `ssh`.

   5. Re-attach jobs, check on progress, get results, etc.

- Run R jobs on HPC

```
cat submit.slurm
```

```
## #!/bin/bash
## #SBATCH --job-name=Sample_Slurm_Job
## #SBATCH --ntasks=1
## #SBATCH --nodes=1
## #SBATCH --mem=1gb
## #SBATCH --time=00:00:20
## #SBATCH --partition=standard
## #SBATCH --account=bios511fa22
##
## cd /xdisk/yiwenliu/netid/linux-basics-II/
##
## module load R
## Rscript arg_meanEst.R seed=1234 'data="poisson"' n=100 > test.output
```

Large simulation studies

R in conjuction with `nohup` or `screen` can be used to orchestrate a large simulation study.

- It can be more elegant, transparent, and robust to parallelize jobs corresponding to different scenarios (e.g., different generative models) outside of the code used to do statistical computation.

- We consider a simulation study in R but the same approach could be used with code written in Julia, Matlab, Python, etc.

- Suppose we have

   - `meanEst.R` (./meanEst.R?_&d2lSessionVal=msHLP93GphIxJVZ4ex9VUCmde&ou=1203592) which runs a simulation based on command line argument `n`.
   - A large collection of `n` values that we want to use in our simulation study.

- Option 1: manually call `meanEst.R` for each setting.

- Option 2: automate calls using R and `nohup`. autoSim.R (./autoSim.R?_&d2lSessionVal=msHLP93GphIxJVZ4ex9VUCmde&ou=1203592)

```
cat autoSim.R
```

```
## # autoSim.R
##
## n.iter = seq(100, 500, by=100)
## for (n in n.iter) {
##   arg = paste("seed=1234", "dist='gaussian'",  "n=", n, sep=" ")
##   out.file = paste("n_", n, ".txt", sep="")
##   sysCall = paste("nohup Rscript meanEst.R ", arg, " > ", out.file)
##   system(sysCall)
##   print(paste("sysCall=", sysCall, sep=""))
## }
```

```
Rscript autoSim.R
```

```
## [1] "sysCall=nohup Rscript meanEst.R  seed=1234 dist='gaussian' n= 100  >  n_100.txt"
## [1] "sysCall=nohup Rscript meanEst.R  seed=1234 dist='gaussian' n= 200  >  n_200.txt"
## [1] "sysCall=nohup Rscript meanEst.R  seed=1234 dist='gaussian' n= 300  >  n_300.txt"
## [1] "sysCall=nohup Rscript meanEst.R  seed=1234 dist='gaussian' n= 400  >  n_400.txt"
## [1] "sysCall=nohup Rscript meanEst.R  seed=1234 dist='gaussian' n= 500  >  n_500.txt"
```

- Now we just need write a script to collect results from the output files.